The Tcl Family of Tools

Editor's Note: Cross references in the text refer to chapters in the companion book, *UNIX: The Complete Reference, Second Edition,* by Rosen, Host, Klee, Farber, and Rosinski.

We have discussed some of the many tools available in the UNIX environment for creating scripts, including the shell, **awk**, and **perl**, in the book. In this chapter we will discuss another important set of tools, the Tcl family. Tcl (pronounced "tickle") is a general-purpose command language, developed in 1987 by John Ousterhout while at the University of California at Berkeley, originally designed to enable users to customize tools or interactive applications by writing Tcl scripts as "wrappers." Although Tcl is ideally suited for this purpose, over the years it has developed into a robust scripting language powerful enough to be used to write tools or applications directly. Tcl is especially useful for building applications that use graphical user interfaces, and it may be coupled with routines that use **perl**. Many web applications are examples of this combination of Tcl with **perl**, as well as just Tcl applications.

Along with the Tcl language itself, several major Tcl applications have been written that extend and complement the functionality of Tcl. Two of the most important Tcl applications are Tk and Expect. *Tk* provides an easy means to create graphical user interfaces based on the X11 toolkit for the X Window System. *Expect* is integrated on top of Tcl and provides additional commands for interacting with applications. This enables users to easily automate many interactive programs so that they do not have to wait to be prompted for input (which cannot be done with shell programming). For example, Expect can be used for "talking" to interactive programs such as **ftp**, **rlogin**, **telnet**, or **tip**. Other extensions to the Tcl/Tk family include XF, which takes the idea of Tk and puts a graphical user interface on it, thereby making it even easier to build graphical user interfaces; Tcl-DP, which provides additional commands to support distributed programming; and Ak, which is an audio extension that provides Tcl commands for playback, recording, telephone control, and synchronization.

In many ways Tcl is analogous to **perl** or a UNIX shell in terms of the types of scripts for which it can be used. Like **perl** or shell, Tcl is also an interpreted language, which means that scripts are written and then directly executed by the Tcl interpreter, with no compilation of the program required. The Tcl language, however, is considerably simpler than **perl**. Moreover, Tcl and its extension applications make a robust functionality available.

Graphical components can be created with Tk, and communications between components and processing of user input can be programmed with Tcl. With the combination of Tcl and Tk it is possible to build fully functional graphical applications. Both of these languages are built as C library packages, so it is possible to integrate C code into Tcl applications if needed.

This is a plausible approach for larger applications or for implementing algorithms where performance is a critical issue. The nice thing is that because Tcl and C are compatible you can build large-scale applications that take advantage of the simplicity and flexibility of Tcl while relying on the structure and high performance of C when necessary.

The tools available within Tcl have been or are being ported to a variety of operating systems. This means that you can use your scripts with minor alterations on many different platforms.

This chapter introduces Tcl, Tk, and Expect, providing enough information to help you get started using these tools to build your own scripts. Refer to the end of this chapter for a listing of FTP sites you can access to obtain these tools and for references that will help you learn more about the Tcl family of tools.

Obtaining Tcl, Tk, and Expect

You can obtain copies of Tcl, Tk, and Expect from various sites on the Internet. In particular, you can obtain Tcl and Tk from *http://www.tcl.tk/.* You can obtain Expect from *http://expect.nist.gov.* For additional information, you can read the latest version of the FAQ in the newsgroup *comp.lang.tcl* or consult the Tcl/Tk web page at *http://www.tcl.tk/.* See the section "How to Find Out More" at the end of this chapter for details.

Tcl

Tcl is a fairly simple *interpreted* programming language usually used for customizing tools and applications, as well as for building your own simple tools and applications. Because the Tcl commands are parsed and executed by an interpreter at run time, it is easy to iteratively create a program. Rather than going through a new compilation step each time you make a change to your program, all you need to do is run the Tcl command interpreter. Many people find it more convenient to work with an interpreted language than a compiled language because it is much quicker to test incremental changes. (Other interpreted languages include **perl**, **awk**, and the shell.) This approach works particularly well for programs of small to medium complexity where run-time performance is not an issue.

Learning the Tcl Language

As a language, Tcl is fairly straightforward and considerably more consistent and compact than **perl**. The Tcl syntax is consistent for all its commands, so the rules for determining how commands get parsed by the Tcl interpreter are easy to follow. Tcl itself contains about 60 built-in commands, and each of the add-on applications in the family adds additional commands.

A Tcl script consists of one or more commands that must be separated by newlines or semicolons. Each command consists of one or more words, separated by spaces or tabs. The Tcl interpreter evaluates commands in two passes. The first pass is a parse of the command to perform substitutions and divide the command into words. Every command is parsed according to the same set of rules, so the behavior of the language is extremely consistent and predictable.

The second pass is the execution pass. This is when Tcl associates meaning to the words in the command and takes actions. The first word of a command is always the command name,

and the Tcl execution pass uses it to call a procedure for processing the remaining words of the command, which are the arguments to the procedure. In general, each procedure has a different set of arguments and a different meaning applied to it. The effect of this format is that the rules for specifying the remaining words in a command after the command name vary.

Getting Started with Tcl

You have two ways to run the Tcl interpreter. One way is to use an interactive shell, **tclsh**, which accepts Tcl commands and outputs the results of these commands. This is handy for testing various Tcl commands to get a feel for how they work. The second way is to create a file that contains a Tcl program and execute it. This is how tools or applications written in Tcl are created and distributed. A Tcl program written in this manner is commonly called a *Tcl script.* Whether you use **tclsh** or create a Tcl script, the language is exactly the same. The only difference is that the first line in a Tcl script consists of the pathname showing where **tclsh** is located on your system so that the shell knows to call that interpreter instead of the shell interpreter. For example (assuming that Tcl is installed on your machine under */usr/local/bin*), you can create a Tcl script by entering the following in a file named *myscript*:

```
#!/usr/local/bin/tclsh
expr 2 + 2
```

Now make *myscript* executable and execute it; **tclsh** will interpret it and return the number 4 as output. Note that the first line of the script tells your shell to call **tclsh** (this is a feature of the shell). The second line contains a command, **expr**, which is a way to tell Tcl to evaluate an expression.

To run Tcl interactively, type **tclsh** (assuming that Tcl is installed on your machine). Because you are using the interactive Tcl shell, you can now enter Tcl commands. For example, the command

```
expr 2 + 2
```

will print the result 4 to the screen. As mentioned earlier, the first word of each Tcl command is always its name, **expr** in this case. This name is used by the Tcl interpreter to select a C procedure that will carry out the function of the command. The remaining words of the command are passed as arguments to the procedure. Each Tcl command returns a result string ("4" in this case). This brings up one of the reasons Tcl is so easy to use—there are no type declarations to variables or command arguments. Tcl automatically determines the type of the variable according to the context of how it is used (as do **awk** and **perl**). For example, the **set** command is used to assign a value to a variable name; the following lines are all legal Tcl commands, where the variables being set are of type integer, real, and string, respectively:

```
set x 44
set y 12.2
set z "hello world"
```

Tcl knows this without requiring the user to explicitly declare the variable type. Results returned by Tcl are always of type string. As with any language, there is a standard set of rules for performing operations involving different types. For example, the command

```
expr $x + $y
```

results in Tcl passing an integer with value 44 and a real with value 12.2 to the **expr** C procedure. This procedure returns a real of value 56.2, which Tcl converts to a string and displays on the screen. As an aside, notice that the $ symbol is used, as it is in shell, when referencing a variable.

Tcl commands can be embedded by using brackets. For example, the command

```
set a [expr $x + $y]
```

results in variable *a* containing the value 56.2. The Tcl interpreter treats all substitutions the same way. It looks at each word, starting from left to right, and sees if a substitution can be made. This is the same whether the word is a command name or a variable name. When the end of the line is reached, it calls the C procedure that corresponds to the command name. If an expression is in brackets, substitutions are still made but the entire bracketed expression is passed as a single argument to the procedure, and this procedure calls another procedure to get the expression evaluated. Although this may sound odd to users not experienced in using a language like this, the result is that, with a little bit of practice, Tcl becomes intuitive. Because all commands adhere to the policies just explained, learning new commands is easy.

Tcl Language Features

The major features of the Tcl language are described in the following sections. For more details consult one of the references listed in the section "How to Find Out More" at the end of the chapter.

Substitutions

Substitution is used to replace a portion of a line with a value. It is important to understand how substitution works in Tcl because almost all Tcl programs rely heavily on substitution. Tcl provides three forms of substitutions. The first form is *variable substitution.* It is denoted by a $ symbol; variable substitution causes the value of the variable to be inserted into a word, as in the earlier example. The second form is *command substitution,* which causes a word to be replaced with the result of a command. The command must be enclosed in brackets. This was also shown in a previous example. Finally, there is *quoting,* which has several different forms. *Backslash substitution* is one form of quoting that allows for formatting such as newlines or tabs, or to bypass the special meaning of characters such as $ or [. Backslash substitution is denoted by use of the \ character immediately preceding the character on which it is operating. For example, the command

```
set a Sale:\nLobster:\ \$12.25\ lb.
```

results in the output

```
Sale: x
Lobster: $12.25 lb.
```

These lines are the value stored within variable *a.* Note that \n causes a line return, \ followed by a single space causes a space to be set within a word, and \ followed by a $ symbol causes the $ symbol to be set as part of the value of the variable instead of causing a substitution.

Backslash substitution is effective for bypassing the meaning of individual special characters or for applying formatting symbols. Double quotation marks and braces can be used instead. Double quotation marks disable word separators; everything within a set of double quotes appears as a single word to the Tcl parser. Braces additionally disable almost all the special characters in Tcl, such as $, and are particularly useful for constructing expressions as command arguments. As an example of double quotation marks, the command

```
set a "Sale:\nLobster: \$12.25 lb."
```

is equivalent to the previous example. Note that the \ characters preceding the spaces are no longer needed. Because brackets are even more powerful, note that the command

```
set a {Sale:\nLobster: \$12.25 lb.}
```

results in the variable *a* being set to the string

```
Sale:\nLobster: \$12.25 lb.
```

which is probably not what was intended. Instead, the equivalent way to use brackets for the desired effect is shown here:

```
set a {Sale:
Lobster: $12.25 lb.}
```

Variables

Tcl variables have a name and a value, both of which are character strings. As you have already seen, the **set** command is used to assign a value to a name. In addition, Tcl provides an **append** command for appending an additional value to an existing variable, an **incr** command for incrementing an integer variable, and an **unset** command for deleting a variable.

Tcl also provides associative arrays, which are lists of elements, each with a name and a value. For example, the sequence of commands

```
set budget (1999) 1289.78
set budget (2000) 2361.22
set budget (2001) 3509.59
```

results in the associative array named budget acquiring three elements named 1999, 2000, and 2001. These elements take on the values 1289.78, 2361.22, and 3509.59 respectively. Array elements are referenced using the $ symbol in the same way regular variables are referenced, for example, $budget(1999).

The Tcl interpreter sets several predefined variables. For instance, *argc* is a variable containing the number of command-line arguments your program is called with, *argv0* is a variable containing the command name your program is executed as, and *argv* is a variable containing the command-line arguments your program is called with. *env* is an associative array containing the environment variables for your program when it is executed. The name of each element in the array is the name of the environment variable. For example, $env(HOME) contains the value of the HOME environment variable.

Expressions

Expressions combine values to create a new value through use of one or more operators. The simplest way to use Tcl expressions is via the **expr** command. This example first applies the – operator to the values 7 and 3 to produce 4 and then applies the * operator to 4 and 2 to produce 8:

```
expr (7-3) * 2
```

The operators supported in Tcl are the same as for ANSI C (discussed in Chapter 24) except that some of the operators also allow for string operands. Tcl evaluates expressions numerically where possible and only uses string operations if the operands do not make sense as a number. When the operands are of different types, Tcl will automatically convert them. When one operand is an integer and the other is a real, Tcl will convert the integer to a real. When one operand is a nonnumeric string and the other a number (integer or real), Tcl will convert the number to a string. The result of an operation is always the same type as its operands except for logical operators, which return 1 for true and 0 for false. Tcl also provides a set of mathematical functions such as **sin**, **log**, and **exp**.

Lists

Lists in Tcl are ordered sets of elements where each element can have any string value. A simple example of a list is an ordered set of string values such as this:

```
apple banana cherry watermelon
```

In this case the list has four elements. *Tcl* provides commands for creating and manipulating lists. We will describe some of the basic commands here.

The lindex Command The **lindex** command extracts an element from a list according to the index argument. Note that the first element is at index 0. For example, the command

```
lindex (apple banana cherry watermelon) 2
```

returns as output the string `"cherry"`.

The concat and list Commands The **concat** and **list** commands are used to combine lists. **concat** takes a set of lists as its arguments and returns a single list; **list** takes the same arguments but returns a list containing the lists. For example, the command

```
concat (apple banana) (cherry watermelon)
```

returns a list that combines the two lists:

```
(apple banana cherry watermelon)
```

whereas the command

```
list (apple banana) (cherry watermelon)
```

returns a list that contains the two lists: ((apple banana) (cherry watermelon)). The distinction between these two commands is important; the list produced by **concat** contains four elements, whereas the list produced by **list** contains two elements.

The llength Command

The **llength** command will return the number of elements in a list, which means that the command

```
llength {list (apple banana)  (cherry watermelon)}
```

returns the value 2, because this is a list containing two lists. On the other hand, the command line

```
llength {concat (apple banana) (cherry watermelon)}
```

returns the value 4.

The linsert Command

The **linsert** command modifies an existing list by inserting one or more elements in an existing list. It takes three arguments: the list to insert within, the position at which to do the insertion, and the elements to be inserted. For example, the command sequence

```
set mylist (a b c d)
insert $mylist 2 x y z
```

returns (a b x y z c d), with the elements being inserted before the location indicated by the index, since the index starts at 0.

The lreplace, lrange, and lappend Commands The **lreplace** command replaces a section of a list with one or more elements. Its arguments are similar to **linsert**. For example, the command sequence

```
set mylist (a b c d)
lreplace $mylist 2 x y z
```

returns (a b x y z), with the existing elements within the list being replaced by the new elements starting at the location indicated by the index. The **lrange** and **lappend** commands are available for extracting a subset of a list and for appending elements to a list.

The lsearch Command The **lsearch** command searches a list and returns the position of the first element that matches a particular pattern. The arguments to this command are the list to search and the pattern to match. For example, the command sequence

```
set mylist (a b c d c)
lsearch $mylist c
```

returns 2, the position of the first element c in the list.

The lsort Command The **lsort** command sorts a list according to alphabetical or numerical order. For example, the command sequence

```
set mylist ( 4 3 2 1 )
lsort -integer $mylist
```

results in the list (1 2 3 4) being returned. You can also provide your own sorting algorithm via the **–command** option.

The split and join Commands The **split** and **join** commands are used to break apart and combine lists by taking advantage of regular separators that delimit the elements in a list. For example, the command sequence

```
set mylist a:b:c:d
split $mylist :
```

returns (a b c d), a list of all the elements delimited by :. Similarly, the command sequence

```
set mylist a b c d
join $mylist :
```

returns (a:b:c:d), a list of all the elements with the : delimiter inserted in between.

Advanced Features

Now that you have learned some of the basics, consider how some more interesting examples of Tcl code can be constructed. For example, the command sequence

```
set mylist ( 1 2 3 4 )
expr [join $mylist +]
```

returns the value 10. This example combines the **expr** and **join** commands along with the rules for command substitution to produce the result. First the Tcl parser applies the **join** operation to *mylist* to produce 1+2+3+4. Then the **expr** operation is applied to 1+2+3+4 to produce the result 10. The square brackets ensure that the result of the **join** operation is what the **expr** command operates on. Without them the code would not produce the desired result.

Controlling Flow

Tcl provides a means of controlling the flow of execution in a script. Tcl's control flow is very similar to that of the C programming language (see Chapter 24) and includes the **if**, **while**, **for**, **foreach**, **switch**, and **eval** commands.

The if Command The **if** command evaluates the expression and then processes a block of code only if the result of the expression is nonzero, as shown here:

```
if ($a > 1){
    set a 1
}
```

From the point of view of the Tcl parser, the **if** command has two arguments. The first is the expression, and the second is the code block to execute if the result of evaluating the expression is nonzero. The **if** command can also have one or more **elseif** clauses and a final **else** clause, just as C does.

The while, for, and foreach Commands Loops can be created using the **while**, **for**, or **foreach** command. The **while** command takes the same two arguments as the **if** command: an expression and a block of code to execute. The **while** command will evaluate the expression and execute the block of code if the expression is nonzero. It will then repeat this process until the expression evaluates to zero. For example, the code

```
set x 0
while {$x < 10}{
    incr x 1
# do processing here
}
```

results in the **incr** command being executed over and over until the value of *x* reaches 10. The **for** command is similar to **while** but provides more explicit loop control. For example, the command

```
for  (set x 0} {$x < 10} {incr x 1}{
# do processing here
}
```

is equivalent to the **while** command. The **foreach** command is an easy way to iterate over all the elements in a list. It takes three arguments: the name of a variable to place each element in, the name of the list to iterate over, and the block of code to execute for each iteration. For example, the command sequence

```
set x 0
set mylist 1 2 3 4
foreach value $mylist{
    incr x $value
}
```

results in the variable *x* being incremented to 1, then 3, then 6, and finally 10.

The break and continue Commands Loops can be terminated prematurely via the **break** and **continue** commands. The **break** command exits the loop and places flow control to the first command line after the loop. The **continue** command terminates the current iteration of the loop and causes the next iteration to begin.

The eval and source Commands

Tcl provides a couple of special commands, **eval** and **source**, which are useful shortcuts to prevent awkward or inefficient code from being written.

The eval Command

The **eval** command is a general-purpose building block that accepts any number of arguments, concatenates them (inserting spaces in between), and then executes the result. This is useful for creating a command as the value of a variable so that it can be stored for execution later in your program. For example, in the command

```
set resetx "set x 0"
```

the variable *init* can be created to store the command **set x 0** so that at some later point in your script you can reset *x* by using the **eval** command on the variable *resetx*:

```
eval $resetx
```

This results in *x* being set to 0. A more advanced use of **eval** is to force an additional level of parsing by the Tcl parser. You can think of this as having a little Tcl script executed within a bigger Tcl script. A simple example of this type of usage is for passing arguments

to another command. Suppose you want to run the **exec** command to remove all files ending in *.tmp* in your current directory. The most obvious approach will not work:

```
exec rm *.tmp
```

This is because the **exec** command does not perform filename expansion. The **glob** command is needed to provide filename expansion, so the following example should work:

```
exec rm [glob *.tmp]
```

However, the **rm** command will fail because the result from **glob** is passed to **rm** as a single argument and so **rm** will think there is only one file whose name is the concatenation of all files ending in *.tmp.* The solution is to use **eval** to force the entire expression to be divided into multiple words and then passed to **eval**:

```
eval exec rm [glob *.tmp]
```

The **eval** command can be used in many creative ways, and it is possible to write some fairly powerful code in a few lines. However, scripts containing **eval** statements can become difficult to debug or for someone else to understand, so use **eval** judiciously.

The source Command The **source** command reads a file and executes the contents of the file as a Tcl script. This is a good mechanism for sharing blocks of Tcl code among different scripts. The return value from **source** will be the return value from the last line executed within the file. This line, for example, results in the contents of the file *input.tcl* being executed as a Tcl script:

```
source input.tcl
```

Procedures

A Tcl procedure is similar in concept to a function in C or a subroutine in **perl**. It is a way to write a block of code so that it can be called as a command from elsewhere within a script.

The proc Command The **proc** command is used to create a procedure in Tcl. The **proc** command takes three arguments: the name of the procedure, the list of arguments that are passed to the procedure, and the block of code that implements the procedure:

```
proc lreverse { mylist } {
    set j [expr [llength $mylist] - 1]
    while ($j >= 0){
        lappend newlist [lindex $mylist $j]
        incr j -1
    }
    return $newlist
}
```

The lreverse Command

The **lreverse** procedure takes a list as an argument and creates another list that reverses the order of the elements from the original list. This new list is returned to the caller of the procedure.

The variables used within a procedure are local variables. The arguments to the procedure are also treated as local variables; a copy of the variables being passed to the procedure is made when the procedure is called, and the procedure operates on this copy. Thus the original variables from the point of view of the caller of the procedure are not affected. The local variables are destroyed when the procedure exits. Here is an example that shows how the **lreverse** procedure can be called:

```
set origlist {a b c d}
set revlist (lreverse origlist)
```

After these lines are executed, **revlist** contains the returned value from **lreverse**, {d c b a}. **origlist** is unmodified, and all the variables inside **lreverse** are destroyed.

Pattern Matching

Tcl has two ways to do pattern matching. The simpler way is "glob" style pattern matching, which is the method for UNIX filename matching used by the shell. This is implemented using the command string match followed by two arguments, the pattern to match and the string to match on. For example, the code

```
set silly (jibber jabber)
foreach item $silly{
    if [string match ji* $item] {
    puts "$item begins with ji"
    }
}
```

results in a match on "jibber" and no match on "jabber."

The regexp Command Tcl also provides more powerful facilities for string manipulation via pattern matching using regular expressions just as the **egrep** program does. Regular expressions are built from basic building blocks called *atoms.* In Tcl the **regexp** command invokes regular expression matching. In its simplest form it takes two arguments: the regular expression pattern, and an input string. The input string is compared against the regular expression. If there is a match, 1 is returned; otherwise, 0 is returned. For example, the command sequence

```
set s "my string to match"
regexp my $s
```

returns 1, whereas the command sequence

```
set s "my string to match"
regexp [A-Z] $s
```

returns 0 (because there are no capital letters in *s*).

The regsub Command The **regsub** command extends the **regexp** idea one step further by allowing substitutions to be made. It takes a third argument, which is the string to substitute for the string that is matched, and a fourth argument, which is the variable in which to store the new string. For example, the command

```
regsub pepper "peter piper picked a pepper" pickle newline
```

results in newline containing the string "peter piper picked a pickle". Because there was a match, 1 will be returned; otherwise, 0 will be returned and the new value of newline will not be created.

The string index, string range, and string length Commands The other string manipulation commands are all based on options of the **string** command. The **string index** command will return the character from a string indicated by the position specified as the index. For example, the command

```
string index "talking about my girl" 11
```

returns u, because the first character is at position 0. The **string range** command returns the substring between the start and stop indices indicated. For example, the command

```
string range "talking about my girl"  14 20
```

returns "my girl", as does the command

```
string range "talking about my girl"  14 end
```

It is also worth mentioning the **string length** command, which returns the number of characters in a string. For example, the command

```
string length "talking about my girl"
```

returns 20.

File Access

The normal UNIX file naming syntax is recognized by Tcl. The commands for file I/O are similar to those for the C language. Here is an example script that illustrates basic file I/O functionality. Type the following lines into a file named *tgrep*:

```
#!/usr/local/bin/Tclsh
if {$argc != 2} {
    error "Usage tgrep pattern filename"
}
set f [open [lindex $argv 1] r]
set pat [lindex $argv 0]
while {[gets $f line] >= 0} {
    if {regexp $pat $line} {
        puts $line
    }
}
close $f
```

This script behaves similarly to the UNIX **grep** program. You can invoke it from the shell with two arguments, a regular expression pattern and a filename, and it will print out the lines in that file that match the regular expression.

Assuming that the correct number of arguments are supplied on the command line, the **tgrep** script will open the file named as the second command-line argument. This file will

be open in read-only mode because the second parameter of the **open** command is *r* (*w* is used for write mode). The **open** command will return a filehandle, which is contained in variable *f*. The variable *pat* is set to contain the first command-line argument, which is the pattern to match. The **gets** command is used to read the next line of the file and store it in the variable line. This is done for each line of the file. **gets** returns 0 after the last line of the file is read, and the **while** loop will exit. Meanwhile, the **regexp** command is used to compare the line read from the file with the pattern to match against, and if there is a match, the **puts** command is used to place the line in an output file, which, in this case, is *stdout* because no filehandle is included as an argument. Finally, note that the file is closed after it is through being accessed, which is good programming technique.

Processes

The **exec** command in Tcl can be used to create a subprocess that will cause your script to wait until the subprocess completes before it continues executing. For example, the command

```
exec date
```

results in the **exec** command executing **date** as a subprocess. Whatever the subprocess writes to standard output is collected by **exec** and returned. In this case, a line is returned that indicates the date when the subprocess was executed.

Pipes can also be constructed using the **open** command in conjunction with the **puts** or **gets** command. This will return an identifier that you can use to transfer data to and from the pipe. **puts** will write data on the pipe, and **gets** will read data from the pipe. For example, the commands

```
set pipe_id [open { | wc} w]
puts pipe_id "eeny meeny miney mo"
```

result in the string "eeny meeny miney mo" being piped to the **wc** program. The result of the **wc** execution will be written to standard out of the Tcl script.

Tcl/TK Plugins for a Web Browser

If you are a web developer, you should know that a Tcl/Tk scripting plug-in is available for your web browser to help you to develop applets (small applications) called *Tclets.* These Tclets can be used in many of the ways Java is used, with the added features of being faster and more easily developed. The Tcl/Tk plugin, currently version 3 for Tcl/Tk 8.*x*, is available through a number of web sites, but the best site to get it from is *http://www.tcl.tk/.* Many of the original Tcl/Tk developers at Sunscript have moved on to work as developers for this site, so this site has a lot of useful tools for and information concerning Tcl/Tk and Tclet applications development on the web.

Tclets

Tclets (pronounced "ticklets") are applets (small programs) that are created in Tcl for use on the web. Although many current web applications are written in Java, Tclets are becoming more popular for use in dynamic applications, since they run faster and can be developed much faster than Java scripts. These applications are developed under the Netscape environment and are a logical growth path from the initial days of Java scripts running on Sun equipment using Netscape as the browser. Currently there is no equivalent

of Tclets in the Microsoft (ActiveX) arena, but Microsoft will assuredly address this lack for developers who use Microsoft Internet Explorer to enhance ActiveX.

If you wish to view some sample Tclets, you may do so by first loading the Tcl/Tk plug-in as specified previously, and then running one of the demos at the Scriptics site.

Tk Basics

Tcl is most popularly used in conjunction with Tk. Tk is an application extension to Tcl that enables you to build graphical user interfaces. It is based on the X11 Window System (discussed in another chapter on this companion web site) but provides a simpler set of commands for programming a user interface than the native X11 toolkit does. It is quite valuable to integrate Tk into your repertoire of programming skills. You then have a means for creating professional-looking graphical user interfaces for the tools and applications you create. And this is valuable for interacting with users because the UNIX command line, although very useful in its own right, is limited in terms of the ways it can interface with users.

To run Tk, use the interactive shell **wish** instead of **tclsh**. Assuming that the Tcl and Tk toolkits are installed on your system, and that you are running under an X Windowing system such as Motif, type **wish** to invoke the Tk windowing shell. This will cause a small, empty window to appear on your screen and the **wish** shell to be ready to accept user input. For example, if you type

```
button .b -text "Hello world!" -command exit
pack .b
```

the window appearance changes to reflect a small button with the words "Hello world!" on it. If you place the mouse pointer on the "Hello world!" text and click the left button, the window disappears and **wish** will exit. The style is similar to Tcl in that command lines are words separated by spaces, with the first word always the command name (**button** and **pack**, respectively, in this case). The command lines are more complicated, however; the order of arguments is important in some cases, and many arguments are optional. The **button** command expects the name of the new window to be the first argument (b in this case). This is followed by two pairs of configuration options—the **–text** option with value "Hello world!" and the **–command** option with value exit. Other configuration options can be included to deal with issues such as sizing, vertical and horizontal spacing, background and foreground, bordering, and color. As a general observation, quotes are used to construct a word that has spaces within it. In this case "Hello world!" is a single word from the point of view of the Tk (and Tcl) interpreter.

Widgets

The basic building block for a Tk graphical user interface is a *widget*. Several different types of widgets are part of the language, such as buttons, frames, labels, listboxes, menus, and scrollbars, to name a few. Each type of widget is called a *class,* and all class commands are of the same structure, as shown in the button command—the first word is the name of the class command, the second word is the name of the widget that is to be created, and the remaining words are pairs of configuration options.

The basic idea behind building a Tk application is to define the widgets that enable the user to specify the input to and get the results from your application. This means that widgets

that are used to collect information from the user (menus, text input fields, buttons) are tied to an action, which could be to execute another script or tool or to execute a subroutine within your own script. And the result of this action would typically cause some output to be displayed to the user. In Tk parlance, these actions are referred to as *event handlers.*

Widgets are created as a hierarchy; for example, a frame may contain two smaller frames—one containing a scrollbar and a text message and the other containing a bitmap image and a label. The name of the widget reflects this hierarchy with a dot used to separate names. Thus, if frame a contains frame b, which contains label c, the name of label c would be a.b.c. To be aware of this hierarchy is very important to the programmer because it directly maps to the display the user will see.

Event Loops

Tk scripts are almost always event driven. An event is recorded by X11 when something "interesting" occurs, such as when a mouse button is pressed or released, a key is pressed or released on the keyboard, or a pointer is moved into or out of a window (usually via a mouse movement). Besides user-driven events, other types of events can occur, such as the expiration of a timer or file-related events. In Tk when an event arrives, it is processed by executing the action that has been bound to the event.

This basic idea of waiting for events and then taking action is known as the *event loop.* While the action binding is being executed, other events are not considered, so there is no danger of causing a new action binding to interfere with one that is already executing. To promote good responsiveness, the action binding is usually designed to be quick or to be interruptible by passing control back to the event loop.

Geometry Manager

Before widgets can appear on your screen, their relationship to other widgets must be defined. Tk contains a geometry manager to control the relative sizes and locations of widgets. The **pack** command shown previously is the most common geometry manager; it deals with the policies for laying out rows and columns of widgets so that they do not overlap. The many options for controlling the geometry manager enable you to build a graphical user interface that has the "look and feel" you require.

Bindings

The **bind** command associates an action to take with an event. Events consist of user inputs such as keystrokes and mouse clicks as well as window changes such as resizing. The action to take is referred to as a *handler* or as an *event handler.* Tk enables you to create a handler for any X event in any window. Here is an example **bind** command:

```
bind .entry CTRL-D {.entry delete insert }
```

The first argument is the pathname to the window where the binding applies (if this is a widget class rather than a pathname, the binding applies to all widgets of that class). The next argument is a sequence of one or more X events. In this case there is a single event—press the D key while pressing the CTRL key. The third argument is the set of actions or handler for the event, which consists of any Tcl script. In this case the .entry widget will be modified to have the character just after the insertion cursor be deleted. This will be invoked by the script each and every time the CTRL-D input is supplied.

Graphical Shell History Tool

This example shows a simple graphical interface that enables a user to save and reinvoke shell commands. Assume that a file named *redo* contains the following script:

```
#!/usr/local/bin/wish -f
set id 0
entry .entry -width 30 -relief sunken -textvariable cmd
pack .entry -padx 1m -pady 1m
bind .entry <Return> {
    set id [expr $id + 1]
    if {$id > 5 } {
        destroy .b[expr $id - 5]
    }
    button .b$id -command "exec <@stdin >@stdout $cmd" -text $cmd
    pack .b$id -fill X
    .b$id invoke
    .entry delete 0 end
    }
```

The **entry** command creates a text entry line called *.entry* (dot entry) that is 30 characters wide and has a sunken appearance. The user input in this line is captured in the *cmd* variable. The **pack** command is used to tell the pack geometry manager how to display the .entry object. A return by the user will activate the binding shown in brackets.

Each return causes a button to be created. Five buttons are created: b1, b2, b3, b4, and b5. These are displayed in a column with the most recently created button displayed at the bottom via the **pack** command. Each button contains the *cmd* value as its text value via the **–text** option. Once *id* exceeds 5, the oldest button (b[expr $id–5]) is destroyed and the newly created button is inserted at the bottom of the column. The result is a list of the five most recent commands being displayed, along with a text entry area to enter a new command. If the user enters a new command, a button will be created for it and displayed. The **invoke** command will cause that button to be selected, resulting in the execution of the command in the window where the Tk script is being run. If the user selects a button instead, it will cause the command displayed in the button to be executed. The last line removes the command from the entry widget so that a new entry can be input by the user.

The Browser Tool

Finally, here is an example of a browser tool. If the following script is contained in a file named *browse* and made executable, it will return a widget listing of the files in the directory in which you run it. If you double-click on a file, this script will open the file for editing using your default editor (if your EDITOR environment variable is set) or the **xedit** editor:

```
#Creates a listbox and scrollbar widget
scrollbar .scroll -command ".list yview"
pack .scroll -side right -fill y
listbox .list -yscroll ".scroll set" -relief raised -geometry 20x20 \
    -setgrid yes
pack .list -side left -fill both -expand yes
wm minsize . 1 1

#fill the listbox with the directory listing
foreach i [exec ls] {
```

```
    .list insert end $I
}

#create bindings
bind .list <Double-Button-1> {
    set i [selection get]
    edit $I
}
bind .list <Control-q> {destroy .}
focus .list

#edit procedure opens an editor on a given file unless the file is a
#directory in which case it will invoke another instance of this script
proc edit {dir file} {
    global env
    if [file isdirectory $file] {
        exec browse $file &
    }
    else {
        if [file isfile $file] {
            if [info exists env{EDITOR}] {

                exec $env(EDITOR) $file &
            }
            else {
                exec xedit $file &
            }
        else {
            puts "$file is not a directory or regular file."
        }
    }
}
```

Going Further

This section has given a very brief introduction to Tk. Tk has many features that we have not covered. To learn more about Tk, locate and read some of the references provided at the end of the chapter, in the section "How to Find Out More." By doing so, you should quickly become adept at using Tk.

Expect

Expect is a language for automatically controlling interactive programs. An *interactive program* is one that prompts the user for information, waits for the user's response, and then takes some action based on the input. Common examples are **ftp**, **rlogin**, **passwd**, and **fsck**. The shell itself is an interactive program. Expect is intended to mimic the user input so that you do not have to sit there and interact with the program. It can save you a lot of time if you find yourself entering the same command over and over into the programs you run. System administrators find themselves in this situation all the time, but regular users also will see opportunities in which Expect can help them.

Expect is written as a Tcl application. This means that it adds some additional commands on top of the full suite of commands that are already available in Tcl. It is named after the main command that has been added, the **Expect** command. Expect was written by Don Libes of the National Institute of Standards and Technology and is fully documented in his book, *Exploring Expect.*

Examples of Expect

The best way to illustrate how Expect is used is to show a number of examples of where it can be used and then to explain the new commands used in each example. For the first example, consider the **passwd** program. This is used when you want to change your password. The program will prompt you for your current password and then ask you to type in your new password twice. Assuming a legal response was input at each step, the interaction looks like this:

```
$passwd
Current password:
New password:
Retype new password:
$password changed for user <user name>
```

This could be automated by an Expect script that takes the old and new passwords as command-line arguments. Assume that the script is called *Expectpwd* and is run from the shell command line as **Expectpwd** <*oldpassword*> <*newpassword*>. The *Expectpwd* script would be written this way:

```
#/usr/local/bin/Expect
spawn passwd
set oldpass [lindex $argv 0]
set newpass [lindex $argv1]
expect "Current password:"
send "$oldpass\r"
expect "New password:"
send "$newpass\r"
expect "Retype new password:"
send "$newpass\r"
```

The **spawn** command causes the **passwd** program to be executed. The **set** commands should be familiar to you from the Tcl section: The variables *oldpass* and *newpass* are created with the values of the first command-line argument *($argv0)* and the second command-line argument *($argv1),* which are the old password and the new password, respectively. *$argv* is a special array automatically created by the Expect interpreter that contains the command-line arguments. The **expect** command waits for the **passwd** program to output the line "Current password:". The Expect interpreter stops and waits until this pattern is matched before continuing. The **send** command sends the line in quotes to the **passwd** program. The **\r** is used to indicate a carriage return, which is a necessary part of the user response for the input to be acted upon.

Although the added value of this particular program is marginal, it leads to a couple of important observations. The first is that a single shell command line is substituted for the user's having to wait to interactively provide the input. The second is that a system

administrator could benefit greatly by taking this approach if she had to set up a few hundred accounts for new users. (Note that when the **passwd** program is run by a system administrator, it does not ask for the old password and it takes the user name as a command-line argument.)

Automating Anonymous FTP

You have already seen the basics of how Expect works: by expecting a set of patterns to match and sending a set of responses to those patterns. This is great for totally automating a task, but sometimes you may need to return control back to the user. For example, the anonymous FTP login process can be automated and then control returned back to the user for inputting the command to retrieve a file:

```
#!/usr/local/bin/Expect
spawn ftp $argv
set timeout 10
expect {
timeout {puts "timed out"; exit}
"connection refused" exit
"unknown host" exit
"Name"
}
send "anonymous\r"
expect "Password:"
send "maja@arch4.att.com\r"
interact
```

After spawning **ftp** to the site included as the argument to the command line when running this script, it makes an FTP connection to that site; you then supply the anonymous login and provide your e-mail as the password (this is done by convention since the anonymous FTP login does not require a real password). Then control is returned to the user via the **interact** command. At this point the user is free to interact with **ftp** as if he or she had manually supplied all the previous steps.

Also shown here is the **timeout** command, which in this case is set for ten seconds. Notice that the first **expect** command contains a series of pattern/action couplets. If no response is received after ten seconds, the timeout pattern is matched and the script writes "timed out" to standard output and exits. The "connection refused" and "unknown host" responses result in the script exiting (remember the response from **ftp** is displayed to the user). And if Name is matched, flow control continues to the rest of the program. The final line of an **expect** command is allowed to have no action associated with the pattern, and so the conventional style is for the command to check for errors and do the action associated with the command in the previous patterns. The final pattern should be for the successful case so that flow control can continue.

Special Variables in Expect

Expect automatically provides a few special variables. The *expect_out* array contains the results of the previous **expect** command. *expect_out* is an associative array that contains some very useful elements. The element *expect_out(0,string)* contains the characters that were last matched. The element *expect_out(buffer)* contains all the matched characters plus

all the characters that came earlier but did not match. When regular expressions contain parentheses, then the elements *expect_out(1,string), expect_out(2,string),* and so on up to *expect_out(9,string)* will contain the string that matches each parenthesized subpattern from left to right. For example, suppose the string "abracadabra" is processed by the following line of Expect code:

```
expect -re "b(.*)c"
```

In this case, *expect_out(0,string)* is set to "brac", *expect(1,string)* is set to "ra", and *expect_out(buffer)* is set to "abrac". The **–re** option to the **expect** command tells it to use regular expression matching.

xterm Example

As a final example, let's look at an Expect script that spawns an **xterm** process and is able to send and receive input to and from it. This example is useful when you want to bring up another window on the user's terminal to report or gather information instead of interrupting the window where the user is running the script. It also can be used to report information to a remote terminal.

You cannot simply type "**spawn** xterm" because **xterm** does not read its input from a terminal interface or standard input. Instead, **xterm** reads input from a network socket. You can tell **xterm** which program to run at the time you start it, but this program will then run inside of the **xterm** that starts and you will no longer be able to control it. One way to be able to run an **xterm** and control it is by spawning it to interact with a terminal interface that you create.

An easy way to do this is to spawn the **xterm** from an existing Expect script. This requires creating a *pseudo-terminal* interface (known as a *pty* interface) to the **xterm** process. The pty can be thought of as sitting between the Expect script and the **xterm** and handling the communications between them. To do this, the pty is organized to have a master interface and a slave interface. The Expect script will have the master interface, and the **xterm** will have the slave interface. Here is the first half of the example:

```
spawn -pty
stty raw -echo < $spawn_out(slave,name)
regexp ".*(.)(.)" $spawn_out(slave,name) junk a b
set xterm $spawn_id
set $xterm_pid (exec xterm -S$a$b$spawn_out(slave,fd) &)
close -slave
```

The option to run an **xterm** under the control of another process requires **xterm** to be run with the **–Sabn** option, where *a* and *b* are the suffix of the pty name and *n* is the pty file descriptor. Fortunately, these are attainable from the associative array *$spawn_out,* which is automatically created by Expect.

First a pty is instantiated without a new process being created, using the **–pty** option to **spawn**. (A pty is always created by the spawn command and normally associated to a process.) The pty has two interfaces—a master and a slave. The element *$spawn_out(slave,name)* contains the name of the slave interface. Because **xterm** requires its interface

to a pty to be in raw mode and echoing disabled, this is done in the second line. The third line picks off the last two characters (the suffix) of the pty name, which are the last two characters of *$spawn_out(slave,name),* and the fourth line runs **xterm** as a background process with the **–Sabn** flag. The element *$spawn_out(slave,fd)* contains the file descriptor for the slave interface to the pty. Finally, the last line closes the slave file descriptor because the slave side of the pty interface is not needed by Expect.

At this point an **xterm** is now running with a pty associated to it that the Expect script has an interface to. The code to communicate to and receive input from the **xterm** is straightforward:

```
spawn $env(SHELL)

interact -u $xterm "X" {
    send -i $xterm "Press return to go away: "
    set timeout -1
    Expect -i $xterm "\r" {
        send -i $xterm "Thanks!\r\n"
        exec kill $xterm_pid
    exit
    }
}
```

First, a shell is spawned so that it can be the "Expect process" that communicates with the xterm process. The **interact** command with the **–u** option causes the interaction to occur between two processes rather than a process and a user. The first process is the one contained in *$spawn_id* (which represents the shell), and the second process is the ID in the **xterm** variable that was set in the first half of the example with the value that represents the pty interface to **xterm**. The "X" argument to the **interact** command is used to indicate the input to look for to break the interact mode. If the user of the **xterm** enters "X" she will then see the string "Press return to go away: " and upon pressing RETURN, she will see the string "Thanks!" and the window will disappear. The **–i** options on the **expect** and **send** commands are used to indicate that the place to be looking is the process identified in **xterm** rather than the default standard I/O. Note that it is sufficient to kill only the **xterm** process because the exiting of the Expect script will cause graceful cleanup of the pty and processes.

Summary

This chapter has provided a brief overview of Tcl, Tk, and Expect. With the information presented here you should be able to get started using these tools to build your own scripts. You should also have the kinds of applications that these tools can be used for. Note that excellent documentation exists for all three tools discussed in this chapter. And there is a thriving and growing community of users, a subset of whom also continue to contribute additional extensions and applications, all of which are freely available on the Internet. As with **perl**, the Tcl family of tools is excellent for developing applications on the web, and they are quite often used together to take advantage of the best features of both when developing a web application.

How to Find Out More

Here are some useful books and resources on Tcl, Tk, Expect, and Perl used together with Tk (Perl is the topic of Chapter 22 in the book):

Ousterhout, John K. *Tcl and the Tk Toolkit.* Reading, MA: Addison-Wesley, 1994.

John Ousterhout wrote the definitive book on Tcl and Tk, but others are helpful as well.

Libes, Don. *Exploring Expect.* Sebastopol, CA: O'Reilly and Associates, Inc., 1995.

In addition to helping you understand Expect, the Libes book is extremely useful for learning about Tcl and Tk.

McMillan, Michael. *PERL from the Ground Up.* Berkeley, CA: McGraw-Hill/Osborne, 1998. This book discusses interfaces between Perl and Tk.

Walsh, Nancy. *Learning Perl/Tk.* Sebastopol, CA: O'Reilly and Associates, 1999.

Welch, B. *Practical Programming in Tcl and Tk, 4th edition.* Upper Saddle River, NJ: Prentice Hall, 2003.

You may find it useful to read the newsgroup *comp.lang.tcl* for more information about Tcl, Tk, Expect, and other extensions of Tcl. In particular, the FAQ in this newsgroup contains a lot of helpful information.

The official web site for Tcl is at *http://www.tcl.tk/.* If you are interested in obtaining the latest version of Tcl, this is the site that gives you all the information about the licensing agreement for tools such as Tcl/Pro and provides the Tck/Tk core (currently 8.4) free of charge. It also provides a wide range of Tcl/Tk applications donated by the Tcl application developer community. If you are interested in obtaining Tcl/Tk, you can also get it from the web site at *http://www.activestate.com/Products/ActiveTcl/.*

The web in general is a good resource for finding out more information about Tcl, Tk, Expect, and other related tools. For example, you might want to look at the Tcl/Tk core development home page, which is at *http://tcl.sourceforge.net/.* From these pages you can find documentation on the tools discussed in this chapter, archive sites for software, and general information about Tcl, Tk, and other tools. You can also look at *http://www.amazon.com/* for an online list of books that cover these topics.